RoleCast: Finding Missing Security Checks When You Do Not Know What Checks Are

Sooel Son
The University of Texas at Austin
samuel@cs.utexas.edu

Kathryn S. McKinley
Microsoft Research and
The University of Texas at Austin
mckinley@cs.utexas.edu

Vitaly Shmatikov
The University of Texas at Austin
shmat@cs.utexas.edu

Abstract

Web applications written in languages such as PHP and JSP are notoriously vulnerable to accidentally omitted authorization checks and other security bugs. Existing techniques that find missing security checks in library and system code assume that (1) security checks can be recognized syntactically and (2) the same pattern of checks applies universally to all programs. These assumptions do not hold for Web applications. Each Web application uses different variables and logic to check the user's permissions. Even within the application, security logic varies based on the user's *role*, *e.g.*, regular users versus administrators.

This paper describes ROLECAST, the first system capable of statically identifying security logic that mediates security-sensitive events (such as database writes) in Web applications, rather than taking a specification of this logic as input. We observe a consistent software engineering pattern—the code that implements distinct user role functionality and its security logic resides in distinct methods and files—and develop a novel algorithm for discovering this pattern in Web applications. Our algorithm partitions the set of file contexts (a coarsening of calling contexts) on which security-sensitive events are control dependent into roles. Roles are based on common functionality and security logic. ROLECAST identifies security-critical variables and applies role-specific variable consistency analysis to find missing security checks. ROLECAST discovered 13 previously unreported, remotely exploitable vulnerabilities in 11 substantial PHP and JSP applications, with only 3 false positives.

This paper demonstrates that (1) accurate inference of application- and role-specific security logic improves the security of Web applications without specifications, and (2) static analysis can discover security logic automatically by exploiting distinctive software engineering features.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithm, Languages, Security

Keywords Security, static analysis, interprocedural analysis, security checks, access control, user roles, PHP, JSP

1. Introduction

By design, Web applications interact with untrusted users and receive untrusted network inputs. Therefore, they must not perform security-sensitive operations unless users hold proper permissions. This mechanism is known as *security mediation*. In Web applications, security mediation is typically implemented by performing *security checks* prior to executing *security-sensitive events*. For example, a program may verify that the user is logged in as an administrator before letting the user update the administrative database.

Our objective in this paper is to develop a robust method for finding missing security checks in Web applications. The main challenge is that each application—and even different roles within the same application, such as administrators and regular users—implements checks in a different, often idiosyncratic way, using different variables to determine whether the user is authorized to perform a particular operation. Finding missing checks is easier if the programmer formally specifies the application's security policy, *e.g.*, via annotations or data-flow assertions [6, 27], but the overwhelming majority of Web applications today are not accompanied by specifications of their intended authorization policies.

Previous techniques for finding missing authorization checks without a programmer-provided policy take the syntactic definition of checks as input. Therefore, they must know *a priori* the syntactic form of every check. For example, Java programs perform security mediation by calling predefined methods in the `SecurityManager` class from

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

the Java libraries [12, 17, 19, 22], and SELinux kernel code calls known protection routines [23]. This approach is suitable for verifying security mediation in library and system code, for which there exists a standard protection paradigm, but it *does not work for finding missing authorization checks in applications* because there is no standard set of checks used by all applications or even within the same application. Whereas analysis of security mediation in libraries requires checks as inputs, analysis of security mediation in Web applications must *infer* the set of role-specific checks from the application's code and produce it as an output.

Our contributions. We designed and implemented ROLECAST, a new static analysis tool for finding missing security checks in Web applications. Given a Web application, ROLECAST automatically infers (1) the set of user roles in this application and (2) the security checks—specific to each role—that must be performed prior to executing security-sensitive events such as database updates. ROLECAST then (3) finds missing security checks. ROLECAST does not rely on programmer annotations or an external specification that indicates the application's intended authorization policy, nor does it assume *a priori* which methods or variables implement security checks.

ROLECAST exploits common software engineering patterns in Web applications. In our experience, these patterns are almost universal. A typical Web application has only a small number of sources for authorization information (*e.g.*, session state, cookies, results of reading the user database). Therefore, all authorization checks involve a conditional branch on variables holding authorization information. Furthermore, individual Web pages function as program modules and each role within the application is implemented by its own set of modules (*i.e.*, pages). Because each page is typically implemented by one or more program files in PHP and JSP applications, the sets of files associated with different user roles are largely disjoint.

Our static analysis has four phases. Phase I performs flow- and context-sensitive interprocedural analysis to collect calling contexts on which security-sensitive events are control dependent. For each context, ROLECAST analyzes interprocedural control dependencies to identify *critical variables*, *i.e.*, variables that control reachability of security-sensitive events. It then uses branch asymmetry to eliminate conditional statements that are unlikely to implement security checks because they do not contain branches corresponding to abnormal exit in the event of a failed check. This step alone is insufficient, however, because many critical variables (*e.g.*, those responsible for logging) are unrelated to security.

Phase II performs role inference. This step is the key new analysis and is critical because different roles within the same application often require different checks. For example, prior to removing an entry from the password database, a photo-sharing application may check the session variable to verify that the user performing the action is logged in with administrator privileges, but this check is not needed for updating the content database with users' photos. ROLECAST infers application-specific roles by analyzing the modular structure of the application. As mentioned above, in PHP and JSP applications this structure is represented by program files. Phase II partitions *file contexts* that use critical variables into *roles*. A file context is simply a coarsened representation of a calling context. The partitioning algorithm arranges file contexts into groups so as to minimize the number of files shared between groups. We call each group a role.

Phase III of ROLECAST determines, within a role, which critical variables are checked *consistently* and classifies this subset of the critical variables as *security-critical variables*. Phase IV then reports potential vulnerabilities in the following cases: (1) if a calling context reaches a security-sensitive event without a check; or (2) if the role contains a single context and thus there is no basis for consistency analysis; or (3) if a check is performed inconsistently (in the majority, but not all calling contexts of the role).

Because our approach infers the Web application's authorization logic under the assumption that the application follows common code design patterns, it may suffer from both false positives and false negatives. This imprecision is inevitable because there is no standard, well-defined protection paradigm for Web applications. Furthermore, no fixed set of operations is syntactically recognizable as security checks (in contrast to prior approaches). Instead, ROLECAST partitions the program into roles and infers, for each role, the security checks and security-relevant program variables by recognizing how they are used consistently (or almost consistently) within the role to control access to security-sensitive events.

We show the generality of our approach by applying it to Web applications written in both PHP and JSP. Without an oracle that finds all missing security checks, we cannot measure the number of false negatives. When evaluated on 11 substantial, real-world PHP and JSP applications, ROLECAST discovered 13 previously unreported security vulnerabilities with only 3 false positives, demonstrating its usefulness for practical security analysis of Web applications.

In summary, this paper demonstrates that it is possible to accurately infer the security logic of Web applications at the level of individual user roles by static analysis, without using any programmer annotations or formally specified policies but relying instead on common software engineering patterns used by application developers.

2. Related Work

Finding security vulnerabilities in Web applications. A lot of research has been devoted to finding and analyzing data-flow vulnerabilities in Web applications [9, 10, 13, 25, 26]. This *taint* analysis focuses on cross-site scripting and SQL injection vulnerabilities, in which untrusted user inputs flow into sensitive database operations without proper sanitation. Missing-check vulnerabilities considered in this paper are *control-flow* vulnerabilities. Consider the following example: `if (user == ADMIN) {DB_query(‘‘DROP TABLE AllUsers’’)}`. Because the query string does not depend on user input, taint analysis will not detect an error if the check is missing. In security parlance, there is an (implicit) information flow from the conditional to the security-sensitive event [7]. The goal of our analysis is not to find such information flows (they are inevitable in all Web applications dealing with user authorization), but to automatically identify the control-flow logic that dominates security-sensitive database operations and check whether or not it is present on all paths leading to these operations.

Balzarotti et al. proposed a method for finding workflow violations caused by unintended entry points in Web applications [1]. Their method focuses on link relations between pages and involves statically analyzing string arguments of page redirect functions. The difficulty of resolving string arguments varies greatly between applications. Their analysis resolved only 78% of string arguments statically. Furthermore, analyzing only inter-page links is not sufficient to discover vulnerabilities caused by missing security checks within the same page. By contrast, the techniques in this paper analyze whole-program control flow of Web applications and are thus more robust.

A promising line of research on finding security bugs in application code requires the programmer to specify an explicit security policy [3, 4, 6, 27]. Few developers today provide such policies for their Web applications. In this paper, we focus on finding missing security checks without explicit policies.

Inferring security policies by static analysis. There are many techniques for (i) inferring security policies implemented by the code, implicitly or with a manual template, and (ii) finding policy violations. Here a *policy* is a mapping from security-sensitive events to security checks that must precede them. For example, Tan et al. use interprocedural analysis to find missing security checks in SELinux [23], while Pistoia et al. [12, 17] and Sistla et al. [19] propose techniques for finding missing security checks in Java library code. These papers rely on policy specification and/or assume that the same policy must hold everywhere for events of a given type.

The problem considered in this paper is significantly harder. In all of the papers cited above, the set of security checks is fixed and known in advance. For example, security checks in the Java class library are calls to methods in the `SecurityManager` class, thus the correct policy must include a call to a `SecurityManager` method prior to executing a security-sensitive event. For instance, networking calls are often protected by `checkConnect` calls. In this case, the problem of policy inference is limited to (i) inferring which of the `SecurityManager` methods implement the correct policy for an event, and (ii) verifying that the policy holds everywhere.

By contrast, there is no uniform protection paradigm for application code. The set of security checks and related program variables varies from application to application and even from role to role within the same application. Therefore, it is not possible to specify, implicitly or explicitly, universal security mediation rules that must hold for all Web applications. Figure 1 shows three instances of security checks from different PHP applications. The checks have no syntactic similarities, rendering previous approaches unusable.

```
<?php
// Authentication check
if(!defined('IN_ADMIN') || !defined('IN_BLOG'))
{
  header('Location: admin.php');
  exit;
}
switch($mode)
{
  case 'edit':
  ...
  // Security-sensitive database operation
  $sql = mysql_query("UPDATE miniblog SET {$sql} WHERE
       post_id = '{$id}'") or die(mysql_error());
  ...
}
?>
```

(a) Miniblog: security logic in `adm/index.php`

```
<?php
...
require_once('./admin.php');
// Authentication check
if ( ! isAdmin() )
  die('You are not the admin.');
$page_title = 'Comment Successfully Deleted';
...
$db = DB_connect($site, $user, $pass);
DB_select_db($database, $db);
...
// Security-sensitive database operation
DB_query("delete from $tblComments where id=$id", $db);
?>
```

(b) Wheatblog: security logic in `admin/delete_comment.php`

```
<?php
session_start();
// Authentication checking routine
if (!$_SESSION['member'])
{
    // not logged in, move to login page
    header('Location: login.php');
    exit;
}
include 'inc/config.php';
include 'inc/conn.php';
...
// Security-sensitive database operation
$q5 = mysql_query("INSERT INTO close_bid(item_name,
      seller_name, bidder_name, close_price) ".$sql5);
$del = mysql_query("delete from dn_bid where dn_name = '"
      .$result['dn_name']."'");
...
?>
```

(c) DNscript: security logic in `accept_bid.php`

Figure 1: Examples of application-specific security logic

Unlike previous approaches, ROLECAST does not rely on domain- or application-specific knowledge of what a security check is. Instead, ROLECAST uses new static analysis algorithms to infer the semantic role that different variables play in the program and automatically identify security checks. It then finds missing checks using role-specific consistency analysis. The idea of analyzing consistency of checks on critical variables was first proposed by Son and Shmatikov [20]. In contrast, this paper is the first to (1) demonstrate the need for *role-specific* analysis because different roles of the same application use different security checks for protection, (2) design and implement a new static method for automatically partitioning contexts into application-specific semantic roles, and (3) exploit the fundamental asymmetry between the branches corresponding to, respectively, successful and failed security checks to help recognize these checks in application code.

Inferring security policies using auxiliary information. Inference and verification of security policies implemented by a given program often benefit from auxiliary information and specifications. For example, Livshits et al. find errors by mining software revision histories [14]. Srivastava et al. use independent implementations of the same Java class library API method to find discrepancies in security policies between implementations [22]. The approach in this paper does not rely on any auxiliary information or specification.

Inferring security policies by dynamic analysis. Dynamic analysis can also infer application- and role-specific policies. For example, Felmetsger et al. collect invariants at the end of each function by dynamically monitoring the application's benign behavior [8]. Bond et al. dynamically analyze calling contexts in which security-sensitive events are executed to create call-chain "signatures," which they use to detect anomalies [2]. Because there is no guarantee that the set of checks observed during test executions is comprehensive, dynamic analysis may miss checks on rarely executed paths. Dynamic and static analyses are complementary. The former is more prone to false negatives, while the latter is more prone to false positives.

3. Security Logic in Web Applications

We focus on server-side Web applications, which are typically implemented in PHP and JSP, as opposed to client-side applications, which are typically implemented in JavaScript. The latter have their own security issues, but are outside the scope of this paper. In this section, we briefly describe PHP and JSP and explain the common design patterns for application- and role-specific security logic.

3.1 PHP and JSP

The PHP (**P**HP: **H**ypertext **P**reprocessor) scripting language is designed for dynamically generating Web pages [15]. PHP is commonly used to implement Web applications with user-generated content or content stored in back-end databases (as opposed to static HTML pages). A recent survey of 120 million domains found that 59% use PHP to generate HTML content [16].

PHP borrows syntax from Perl and C. In PHP programs, executable statements responsible for generating content are mixed with XML and HTML tags. PHP provides basic data types, a dynamic typing system, rich support for string operations, some object-oriented features, and automatic memory management with garbage collection. Instead of a module or class system, PHP programs use a flat file structure with a designated main entry point. Consequently, (1) a network user can directly invoke any PHP file by providing its name as part of the URL, and (2) if the file contains executable code outside of function definitions, this code will be executed. These two features of PHP require defensive programming of each entry point and are a source of security errors. Security analysis of PHP code must consider that every file comprising an application is an alternative entry point.

JSP (**J**ava **S**erver **P**ages) is a Java technology for dynamically generating HTML pages [11]. It adds scripting support to Java and mixes Java statements with XML and HTML tags. Scripting features include libraries of page templates and an expression language. JSP is dynamically typed, but because it builds on Java, it has more object-oriented features than PHP. JSP executes in a Java Virtual Machine (JVM).

Although the languages are quite different, JSP was developed with the same goals as PHP. To demonstrate that our approach provides a generic method for analyzing security of Web applications regardless of the implementation language, we apply our analysis to both JSP and PHP applications. We find that programming practices and application security abstractions are quite similar in PHP and JSP applications, and our analysis works well on real-world programs implemented in either language.

3.2 Translating Web Applications into Java

Translating scripting languages into Java is becoming a popular approach because it helps improve performance by taking advantage of mature JVM compilers and garbage collectors. We exploit this practice by (1) converting Web applications into Java class files, and (2) extending the Soot static analysis framework for Java programs [21] with new algorithms for static security analysis of Web applications.

To translate JSP and PHP programs into Java class files we use, respectively, the Tomcat Web server [24] and Quercus compiler [18]. Tomcat produces well-formed Java; Quercus does not. PHP is a dynamically typed language and the target of every callsite is potentially bound at runtime. Instead of analyzing calls in the PHP code, Quercus translates each PHP function into a Java class that contains a main method and methods that initialize the global hash table and member variables. Every function call is translated by Quercus into a reflective method call or a lookup in the hash table. This process obscures the call graph.

Because our security analysis requires a precise call graph, we must reverse-engineer this translation. We resolve the targets of indirect method calls produced by Quercus using a flow- and context-insensitive intraprocedural symbol propagation. This analysis is described in the appendix.

3.3 Application-Specific Security Logic

Our security analysis targets interactive Web applications such as blogs, e-commerce programs, and user content management systems. Interactive applications of this type constitute the overwhelming majority of real-world Web applications. Since the main purpose of these applications is to display, manage, and/or update information stored in a back-end database(s), access control on database operations is critical to the integrity of the application's state.

Security-sensitive events. We consider all operations that may affect the *integrity* of database to be *security-sensitive events*. These include all queries that *insert*, *delete*, or *update* the database. Web applications typically use SQL to interact with the back-end database. Therefore, ROLECAST marks INSERT, DELETE, and UPDATE `mysql_query` statements in PHP code as security-sensitive events. Note that statically determining the type of a SQL query in a given statement requires program analysis. ROLECAST conservatively marks all statically unresolved SQL queries as sensitive. For JSP, ROLECAST marks `java.sql.Statement.executeQuery` and `.executeUpdate` calls executing INSERT, DELETE, or UPDATE SQL queries as security-sensitive events.

We deliberately do not include SELECT and SHOW queries which retrieve information from the database in our definition of security-sensitive events. Many Web applications intend certain SELECT operations to be reachable without any prior access-control checks. For example, during authentication, a SELECT statement may retrieve a stored password from the database in order to compare it with the password typed by the user. Without a programmer-provided annotation or specification, it is not possible to separate SELECT operations that need to be protected from those that may be legitimately accessed without any checks. To avoid generating a prohibitive number of false positives, we omit SELECT and SHOW operations from our analysis of Web applications' security logic.

Examples of security logic. Figure 1 shows the security logic of three sample Web applications: Miniblog, Wheatblog, and DNscript. DNscript is a trading application for sellers and buyers of domain names. Figure 1(a) shows a security-sensitive event on line 13. This database operation is only executed if `IN_ADMIN` and `IN_BLOG` are defined (line 3), otherwise the program exits immediately. Figure 1(b) shows a security-sensitive event on line 10. It is executed only when the return value of `isAdmin()` is true; otherwise the program exits by executing the `die` command. Figure 1(c) shows two security-sensitive events (lines 14 and 15). They are executed only if the user is logged in and the session authentication flag in `$_SESSION['member']` is set. By studying these and other examples, we found several robust design patterns for security logic. These patterns guide our analysis.

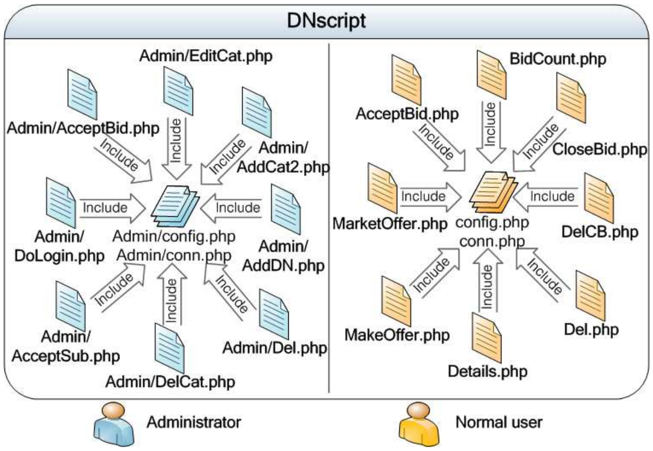

Figure 2: File structure of DNscript

Security logic as a software design pattern. To identify application- and role-specific security logic, we take advantage of the software engineering patterns commonly used by the developers of Web applications. A Web application typically produces multiple HTML pages and generates each page by invoking code from several files. The following three observations guide our analysis.

Our first observation about security logic is that *when a security check fails, the program quickly terminates or restarts*. Intuitively, when a user does not hold the appropriate permissions or his credentials do not pass verification, the program exits quickly.

Our second observation about correct security logic is that *every path leading to a security-sensitive event from any program entry point must contain a security check*. This observation alone, however, is not sufficient to identify checks in application code because different paths may involve different checks and different program variables.

Our third observation is that *distinct application-specific roles usually involve different program files*. Since the main purpose of interactive Web applications is to manage user-specific content and to provide services to users, users' privileges and semantic roles determine the services that are available to them. Therefore, the application's file structure, which in Web applications represents the module structure, reflects a clear distinction between roles defined by the user's privileges. For instance, blog applications typically include administrator pages that modify content and register new user profiles. On the other hand, regular blog users may only read other users' content, add comments, and update their own content. In theory, developers could structure their applications so that one file handles multiple user roles, but this is not the case in real-world applications.

In the Web applications that we examined, individual program files contained only code specific to a single user role. Figure 2 shows a representative example with a simple page structure taken from the `DNscript` application. DNscript supports two types of users: an administrator and a regular user. All administrator code and pages are in one set of files, while all user code and pages are in a different set of files.

Our analysis exploits the observation that security checks within each role are usually very similar. Inferring the roles requires automatic partitioning of contexts based on commonalities in their security logic. This component is the most challenging part of our analysis.

3.4 Example Vulnerability

Because every application file is also a program entry point, PHP and JSP developers must replicate the application's security logic in every file. In particular, even if the code contained in some file is intended to be called only from other files, it must still be programmed defensively because a user may invoke it directly via its URL. Adding these security checks is a repetitive, error-prone task and a common cause of missing-check vulnerabilities.

Figure 3 shows an example of a security vulnerability due to a missing check on a path from an unintended program entry point. The intended entry point index.php correctly checks permissions for security-sensitive database operations. However, a malicious user can directly invoke delete_post.php by supplying its URL to the Web server and execute a DELETE query because delete_post.php is missing a check on the `$_SESSION` variable. In complex applications, the number of unintended entry points can be very large. This greatly complicates manual inspection and motivates the need for automated analysis.

4. Analysis Overview

RoleCast has four analysis phases. Phase I identifies *critical* variables that control whether security-sensitive events execute or not. Phase II partitions contexts into groups that approximate application-specific user roles. For each role, Phase III computes the subset of critical variables responsible for enforcing the security logic of that role. Phase IV discovers missing security checks by verifying whether the relevant variables are checked consistently within the role.

To identify critical variables, Phase I performs interprocedural, flow- and context-sensitive control-dependence and data-flow analysis. It refines the set of critical variables using branch asymmetry, based on the observation that failed authorization checks quickly lead to program exit. To infer application roles, Phase II maps the set of methods responsible for checking critical variables to program files and partitions them into groups, minimizing the number of shared files between groups. This algorithm seeks to discover the popular program structure in which developers put the code responsible for different application roles into different files. This heuristic is the key new component of our analysis and works well in practice.

index.php

```
// Security check
if( ! $_SESSION[‘logged_in’] )
{
  doLogin();
  die;
}
if( isset($_GET['action']) )
  $action = $_GET['action'];
  switch( $action ){
  case 'delete_post':
    include 'delete_post.php';
    break;
  case 'update_post':
    include 'update_post.php';
    break;
    .....
  default:
    include 'default.php';
}
```

delete_post.php

```
// No security check
if (isset($_GET['post_id']))
  $post_id = $_GET['post_id'];
DBConnect();
// Security-sensitive event
$sql="DELETE FROM blogdata WHERE post_id=$post_id";
$ret=mysql_query($sql) or die("Cannot query the
     database.<br>");
Ln6: .....
```

update_post.php

```
// Security check
if(!$_SESSION['logged_in']) die;
if (isset($_GET['post_id']))
  $post_id = $_GET['post_id'];
if (isset($_GET['content']))
  $content = $_GET['content'];
DBConnect();
// Security-sensitive event
$sql = "UPDATE table_post SET cont=$content WHERE id=
     $post_id";
$ret=mysql_query($sql) or die("Cannot query the
     database.<br>");
....
```

Figure 3: Example of a missing security check

Phase III considers each role and computes the subset of critical variables that are used *consistently*—that is, in a sufficiently large fraction of contexts associated with this role—to control reachability of security-sensitive events in that role. The threshold is a parameter of the system. Phase IV reports a potential vulnerability whenever it finds a security-sensitive event that can be reached without checking the security-critical variables specific to the role. It also reports all roles that involve a single context and thus preclude consistency analysis, but this case is relatively rare.

5. Phase I: Finding Security-Sensitive Events, Calling Contexts, and Critical Variables

Our algorithm for identifying security logic takes advantage of the following observations:

1. Any security check involves a branch statement on one or more *critical* variables.
2. In the branch corresponding to the failed check, the program does not reach the security-sensitive event and exits abnormally. For example, the program calls `exit`, calls `die`, or returns to the initial page.
3. The number of program statements in the branch from the check to the abnormal exit is significantly smaller than the number of program statements in the branch leading to the security-sensitive event.
4. Correct security logic must consistently check a certain subset of critical variables prior to executing security-sensitive events.

This section describes our algorithms that, for each security-sensitive event, statically compute the following: calling contexts, critical branches (*i.e.*, conditional statements that determine whether or not the security-sensitive event executes), critical methods (*i.e.*, methods that contain critical branches), and critical variables (*i.e.*, variables referenced by critical branches).

Our analysis is fairly coarse. It only computes *which* variables are checked prior to security-sensitive events, but not *how* they are checked. Therefore, RoleCast will miss vulnerabilities caused by incorrectly implemented (as opposed to entirely missing) checks on the right variables.

5.1 Security-Sensitive Events and Calling Contexts

Our analysis starts by identifying security-sensitive operations that may affect the integrity of the database. A typical Web application specifies database operations using a string parameter passed to a generic SQL query statement. We identify all calls to `mysql_query` in PHP and `java.sql.Statement.executeQuery` and `java.sql.Statement.executeUpdate` in JSP as candidates for security-sensitive events. The same call, however, may execute different database operations depending on the value of its string parameter. Therefore, we perform an imprecise context-sensitive data-flow analysis to resolve the string arguments of database calls and eliminate all database operations that do not modify the database (see Section 3.3) from our set of security-sensitive events.

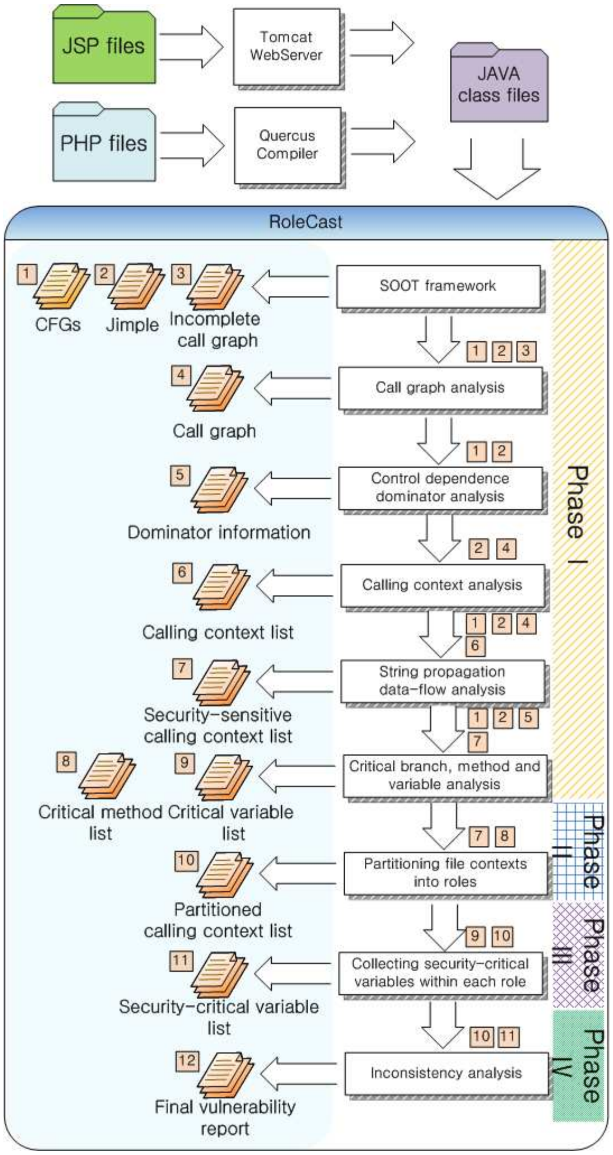

Figure 4: Architecture of RoleCast.

RoleCast computes the set of all calling contexts for each security-sensitive event e. RoleCast identifies the methods that may directly invoke e, then performs a backward depth-first pass from each such method over the call graph. (The call graph construction algorithm is described in the appendix.) The analysis builds a tree of contexts whose root is e and whose leaves are program entry points. For each calling context cc corresponding to a call-chain path from e to a leaf, the (cc, e) pair is added to the set of all calling contexts. This analysis records each invoked method only once per calling context, even in the presence of cyclic contexts. This is sufficient for determining whether or not a security check is present in the context.

Next, RoleCast propagates the strings passed as parameters in each calling context cc to the candidate event e. The goal is to eliminate all pairs (cc, e) where we can statically prove that e cannot be a security-sensitive event, *i.e.*, none of the statically feasible database operations at e affect the integrity of the database because they can only execute SELECT or SHOW queries.

We find that many Web applications generate SQL queries by assigning a seed string constant to a variable and then concatenating additional string constants. The seed string usually identifies the type of the SQL query (UPDATE, SELECT, *etc.*). Therefore, ROLECAST models string concatenation, assignment of strings, and the behavior of string `get()` and `set()` methods. It performs forward, interprocedural, context-sensitive constant propagation on the string arguments for each calling context cc of event e. ROLECAST does not model the value of strings returned by method calls not in the calling context. If the string is passed as an argument to some method $m \notin cc$, we conservatively assume that the string is modified and ROLECAST marks event e as security-sensitive. Otherwise, the analysis propagates string values of actual arguments to the formal parameters of methods.

If this analysis proves that e is a SELECT or SHOW query, then ROLECAST removes the (cc, e) from the set of calling contexts. The "unresolved" column in Table 1, explained in more detail in Section 9, shows that fewer than 5% of query types are unresolved, while for at least 95% of all database operations in our sample Web applications, ROLECAST successfully resolves whether or not they are sensitive, *i.e.*, whether they can affect the integrity of the database.

5.2 Critical Branches and Critical Methods

For each calling context and security-sensitive event pair (cc, e), ROLECAST performs an interprocedural control-dependence analysis to find the *critical branches* $B(cc, e)$ performed in the *critical methods* $CM(cc, e)$. These branches determine whether or not e executes. A critical method contains one or more critical branches on which e is interprocedurally control dependent. Note that some critical methods are in cc and some are not. For the reader's convenience, we review the classical intraprocedural definition of *control dependence* [5].

DEFINITION 1. *If* $G = (N, E)$ *is a control-flow graph (CFG) and* $s, b \in N$, $b \rightsquigarrow s$ *iff there exists at least one path reaching from* b *to* s *in* G.

DEFINITION 2. *If* $G = (N, E)$ *is a control-flow graph (CFG) and* $s, b \in N$, s *is* **control dependent** *on* b *iff* $b \rightsquigarrow s$ *and* s *post-dominates all* $v \neq b$ *on* $b \rightsquigarrow s$, *and* s *does not post-dominate* b.

The set of branch statements on which e is control dependent is computed in two steps.

1. ROLECAST uses intraprocedural control dependence to identify branch statements in methods from cc that control whether e executes or not. For each method $m_i \in cc$ where m_i calls m_{i+1}, the algorithm finds branch statements on which the callsite of m_{i+1} is control dependent and adds them to $B(cc, e)$.
2. ROLECAST then considers the set of methods N such that the callsite of $n_i \in N$ dominates some method $m \in cc$ or n_i is called unconditionally from $n_j \in N$. Because every method $n_i \in N$ is invoked before reaching e, n_i *interprocedurally dominates* e. For each $n_i \in N$, ROLECAST finds branch statements on which the program-exit calls in n_i (if any) are control dependent and adds them to $B(cc, e)$.

Next, ROLECAST eliminates statements from B that do not match our observation that failed security checks in Web applications terminate or restart the program quickly. To find branch statements in which one branch exits quickly while the other executes many more statements, ROLECAST calculates the *asymmetric ratio* for each $b \in B$ as follows.

ROLECAST counts the number of statements in, respectively, the shortest path reaching program termination and the shortest path reaching e. Each statement in a loop counts as one statement. The asymmetric ratio is the latter count divided by the former. The larger the value, the more asymmetric the branches are. If the calculated asymmetric ratio for b is less than the threshold θ_{asymm}, we remove b from B because b does not have a branch that causes the program to exit quickly and thus is not likely to be security-critical. Our experiments use 100 as the default θ_{asymm} threshold when the calculated ratio for one or more branch statements is greater than 100; otherwise, we use the median ratio of all branch statements. As Table 3 shows, the results are not very sensitive to the default value. In our sample applications, applying the default filter reduces the number of critical branches by 36% to 83% (54% on average).

After this step, the set $B(cc, e)$ contains critical branch statements. We map $B(cc, e)$ to the set of critical methods $CM(cc, e)$ that contain one or more branches from $B(cc, e)$. Recall that some critical methods are in cc and some are not. Critical methods are a *superset* of the methods responsible for implementing the application's security logic.

5.3 Critical Variables

Informally, a program variable is *critical* if its value determines whether or not some security-sensitive event is reached. All *security-critical* variables involved in the program's security logic (*e.g.*, variables holding user permissions, session state, *etc.*) are critical, but the reverse is not always true: critical variables are a superset of security-critical variables. We derive the set of critical variables from the variables referenced directly or indirectly by the critical branches $B(cc, e)$. Section 8 further refines the set of critical variables into the set of security-critical variables.

Given $B(cc, e)$, we compute the set of critical variables $V(cc, e)$ where $v \in V$ iff $\exists b \in B$ that references v directly or indirectly through an intraprocedural data-flow chain. We use a simple reaching definitions algorithm to compute indirect references within a method. We compute the backward intraprocedural data-flow slice of v for all v referenced by b. Thus if b references v and v depends on u (*e.g.*, $v = foo(u)$), we add v and u to V.

We found that intraprocedural analysis was sufficient for our applications, but more sophisticated and object-oriented applications may require interprocedural slicing.

6. Phase II: Partitioning Into Roles

This section describes how ROLECAST partitions applications into roles. We use role partitioning to answer the question: "Which critical variables should be checked before invoking a security-sensitive event e in a given calling context?"

Without role partitioning, critical variables are not very useful because there are a lot of them and they are not always checked before every security-sensitive event. Reporting a vulnerability whenever some critical variable is checked inconsistently results in many false positives (see Section 9). Role partitioning exploits the observation made in Section 3.3 that Web applications are organized around distinct user roles (*e.g.*, administrator and regular user). We note that (1) applications place the code that generates pages for different roles into different files, and, furthermore, (2) the files containing the security logic for a given role are distinct from the files containing the security logic for other roles. These observations motivate an approach that focuses on finding sets of (cc, e) in which the critical methods $CM(cc, e)$ use the same *files* to enforce their security logic.

ROLECAST starts by coarsening its program representation from methods and calling contexts to files. This analysis maps each set of critical methods $CM(cc, e)$ to a set we call the *critical-file context* CF. A file $cf \in CF(cc, e)$ if cf defines any method $m \in CM(cc, e)$. We also define the *file context* $F(cc, e)$. A file $f \in F(cc, e)$ if f defines any method m which interprocedurally dominates e. F is a superset of CF—some files in $F(cc, e)$ are critical and some are not. We refer to the set of all file contexts F of all security-sensitive events in the program as $\widehat{F}$ and to the set of all critical-file contexts CF as $\widehat{CF}$.

Since we do not know *a priori* which file corresponds to which application role, our algorithm generates candidate partitions of $\widehat{CF}$ and picks the one that minimizes the number of shared files between roles. The goal is to group similar critical-file contexts into the same element of the partition. We consider two critical-file contexts similar if they share critical files. To generate a candidate partition, the algorithm chooses a "seed" critical file cf_1 and puts all critical-file contexts from $\widehat{CF}$ that reference cf_1 into the same group, chooses another critical file cf_2 and puts the remaining critical-file contexts from $\widehat{CF}$ that contain cf_2 into another group, and so on. The resulting partition depends on the order in which seed critical files are chosen. Our algorithm explores all orders, but only considers frequently occurring critical files. In practice, the number of such files is small, thus generating candidate partitions based on all possible orderings of seed files is feasible.

To choose the best partition from the candidates, our algorithm evaluates how well the candidates separate the more general file contexts $\widehat{F}$. The insight here is that since programmers organize the *entire* application by roles (not just the parts responsible for the security logic), the correct partition of security-related file contexts should also provide a good partition of all file contexts. ROLECAST thus prefers the partition in which the groups are most self-contained, *i.e.*, do not reference many files used by other groups.

More formally, ROLECAST's partitioning algorithm consists of the following five steps.

1. For each $CM(cc, e)$, compute the critical-file context $CF(cc, e)$ and file context $F(cc, e)$.
2. Eliminate critical files that are common to all $CF(cc, e)$, *i.e.*, if file f belongs to the critical-file context cf for all $cf \in \widehat{CF}$, then remove f from all contexts in $\widehat{CF}$. Since these files occur in every critical-file context, they will not help differentiate roles.
3. Extract a set of *seed files* (SD) from $\widehat{CF}$. We put file f into SD if it occurs in at least the θ_{seed} fraction of critical-file contexts $cf \in \widehat{CF}$. In our experiments, we set $\theta_{seed} = 0.2$. We use only relatively common critical files as the seeds of the partitioning algorithm, which helps make the following steps more efficient.
4. Generate all ordered permutations SD_i. For each SD_i, generate a partition $P_i = \{G_1, \ldots G_k\}$ of $\widehat{CF}$ as follows. Let $SD_i = \{f_1, \ldots f_n\}$. Let $\widehat{TF} = \widehat{CF}$ and set $k = 1$. For $i = 1$ to n, let $C_i \subseteq \widehat{TF}$ be the set of all critical-file contexts from $\widehat{TF}$ containing file f_i. If C_i is not empty, remove these contexts from $\widehat{TF}$, add them to group G_k, and increment k.
5. Given candidate partitions, choose the one that minimizes the overlap between files from different groups. Our algorithm evaluates each candidate $\{G_1, \ldots G_k\}$ as follows. First, for each group of critical-file contexts G_j, take the corresponding set of file contexts F_j. Then, for each pair F_k, F_l where $k \neq l$, calculate the number of files they have in common: $|F_k \cap F_l|$. The algorithm chooses the partition with the smallest $\sum_{k<l} |F_k \cap F_l|$.

Figure 5 gives an example of the partitioning process. At the top, we show five initial file contexts and critical-file contexts produced by Step 1. Step 2 removes the files common to all critical-file contexts, common.php in this example, producing three files, process.php, user.php and admin.php. Step 3 selects these three files as the seed files, because they appear in 1/5, 3/5, and 2/5 of all critical-file contexts, respectively. There are 6 permutations of this set, thus Step 4 produces 6 candidate partitions. In Figure 5, we compare two of them: (admin.php, user.php, process.php)

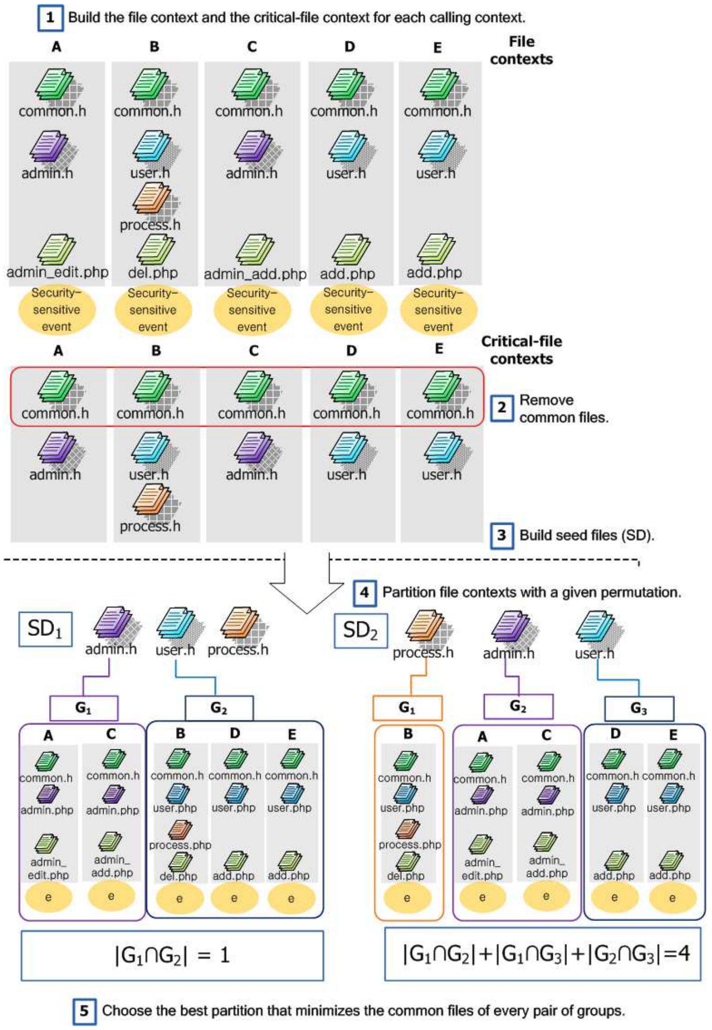

Figure 5: Example of partitioning contexts

and (process.php, admin.php, user.php). The corresponding candidates are $P_1 = \{(A, C), (B, D, E)\}$ and $P_2 = \{(B), (A, C), (D, E)\}$. To decide which one best approximates user roles, Step 5 computes the intersection of file contexts between every pair of groups in each partition and selects the partition with the smallest intersection. In our example, since P_1 has the fewest files (one) in the intersection, RoleCast chooses it as the best partition.

The accuracy of the partitioning step depends on using good seed files. We select files that contain critical variables and appear in many critical-file contexts, based on our observation that Web applications use common security logic in most contexts that belong to the same role. One concern is that exploring all permutations of the seed-file set is exponential in the number of seed files. The selection criteria for seed files are fairly stringent, and therefore their actual number tends to be small. It never exceeded four in our experiments. If the number of seed files grows large, the partitioning algorithm could explore fewer options. For example, it could prioritize by how often a seed file occurs in the file contexts.

Algorithm 1: Partitioning file contexts into roles

```
$P \leftarrow \phi$ { initialize partition candidate set }
$n \leftarrow |SD|$ { get the size of $SD$ }
**for** each $SD_i \in$ n permutation of $SD$ **do**
    $\widehat{TF} \leftarrow \widehat{CF}$ { initialize work list with all critical-file contexts }
    $k \leftarrow 1$
    **for** each $f_j \in SD_i$ **do**
        { put all critical-file context that contain seed file $f_j$ into $G_k$ }
        **for** each $CF \in \widehat{TF}$ **do**
            **if** $f_j \in CF$ **then**
                $G_k \leftarrow G_k \cup CF$
                $\widehat{TF} \leftarrow \widehat{TF} - \{CF\}$
            **end if**
        **end for**
        **if** $G_k$ is not empty **then**
            $k \leftarrow k + 1$ { increment the group index }
        **end if**
        **if** $\widehat{TF}$ is empty **then**
            break { if work list is empty, then break }
        **end if**
    **end for**
    $P_i \leftarrow \{G_1 \ldots G_k\}$ { store the current groups into $P_i$ }
    $DS_i \leftarrow 0$
    { compute the number of common files among all pairs in $G_i$ }
    **for** each pair $(G_a, G_b)_{a<b}$ from $\{G_1 \ldots G_k\}$ **do**
        $F_a =$ the file contexts corresponding to critical-file contexts in $G_a$
        $F_b =$ the file contexts corresponding to critical-file contexts in $G_b$
        $DS_i \leftarrow DS_i + |F_a \cap F_b|$
    **end for**
    $P \leftarrow P \cup \{P_i\}$ { add the current partition to the candidate list }
    **for** $x = 1$ to $k$ **do**
        $G_x \leftarrow \phi$ { reset groups for the next candidate $SD_{i+1}$ }
    **end for**
**end for**
Pick the best $P_i \in P$ that has a minimum $DS_i$
```

7. Phase III: Finding Security-Critical Variables

The groups in the partition computed by Phase II approximate semantic user roles in the application. Phase III computes the *security-critical variables* for each role: the subset of the critical variables that enforce the role's security logic.

We assume that the application correctly checks the security-critical variables in at least some fraction of the critical-file contexts and use this observation to separate security-critical variables from the rest of the critical variables. Recall that there is a one-to-one mapping between each critical-file context $CF(cc, e)$ and the set of its critical variables $V(cc, e)$. Given the partition $\{G_1, \ldots, G_k\}$ of $\widehat{CF}$, let V_i be the set of all critical variables from the critical-file contexts in G_i. We initialize the set of security-critical variables $SV_i = V_i$. RoleCast then removes all variables $v \in SV_i$ that appear in fewer than a $\theta_{consistent}$ fraction of the critical-file contexts in G_i. We use $\theta_{consistent}$ = 0.5 as our default. Table 5 shows that our results are not very sensitive to this parameter. We define the remaining subset SV_i to be the security-critical variables for role i.

				DB operations (|contexts|)			critical branches	
Web applications	**LoC**	**Java LoC**	**analysis time**	**candidates**	**sensitive**	**unresolved**	**candidates**	**asymm**
minibloggie 1.1	2287	5395	47 sec	13	3	0	12	7
DNscript	3150	11186	47 sec	99	26	0	27	5
mybloggie 1.0.0	8874	26958	74 min	195	26	0	135	58
FreeWebShop 2.2.9	8613	28406	110 min	699	175	0	186	82
Wheatblog 1.1	4032	11959	2 min	111	30	0	31	20
phpnews 1.3.0	6037	13086	166 min	80	14	3	65	15
Blog199j 1.9.9	8627	18749	75 min	195	68	2	104	54
eBlog 1.7	13862	24361	410 min	677	261	0	136	17
kaibb 1.0.2	4542	21062	197 min	676	160	0	306	152
JsForum (JSP) 0.1	4242	4242	52 sec	60	32	0	6	1
JSPblog (JSP) 0.2	987	987	16 sec	6	3	0	0	0

Table 1: Benchmarks and analysis characterization

8. Phase IV: Finding Missing Security Checks

This phase reports vulnerabilities. Within each role, all contexts should consistently check the same security-critical variables before performing a security-sensitive event e. Formally, given $sv \in SV_i$, RoleCast examines every (cc, e) in the group of contexts corresponding to role i and verifies whether $sv \in V(cc, e)$ or not. If $sv \notin V(cc, e)$, RoleCast reports a potential security vulnerability. To help developers, RoleCast also reports the file(s) that contains the security-sensitive event e for the vulnerable (cc, e).

RoleCast reports a potential security vulnerability in two additional cases: (1) for each file that executes a security-sensitive event and does not check any critical variables whatsoever, and (2) for each singleton critical-file context (*i.e.*, a role with only one critical-file context). Because there is nothing with which to compare this context, RoleCast cannot apply consistency analysis and conservatively signals a potential vulnerability.

9. Experimental Evaluation

We evaluated RoleCast by applying it to a representative set of open-source PHP and JSP applications. All experiments in this section were performed on a Pentium(R) D 3GHZ with 2G of RAM.

Table 1 shows the benchmarks, lines of original source code, lines of Java source code produced by translation, and analysis time. We have not tuned our analysis for performance. For example, we unnecessarily re-analyze every context, even if we have analyzed a similar context before. Memoizing analysis results and other optimizations are likely to reduce analysis time. The three columns in the middle of the table show the number of contexts for all database operations (candidate security-sensitive events), operations that can affect the integrity of the database (security-sensitive events), and database operations whose type could not be resolved by our analysis. Comparing these three columns shows that our string propagation is effective at resolving the type of database operations and rarely has to assume an operation is security-sensitive because it could not resolve the string argument determining its type. The last two columns show the number of critical branch statements before and after eliminating statements that are not sufficiently asymmetric.

Table 2 shows the results of applying RoleCast to our benchmarks. As described in Section 6, RoleCast partitions calling contexts containing security-sensitive events into groups approximating application-specific user roles. For each role, RoleCast finds critical variables that are checked in at least the $\theta_{consistency}$ fraction of the contexts in this role. If such a critical variable is not checked in one of the contexts, RoleCast reports a potential vulnerability. RoleCast also reports a potential vulnerability if a security-sensitive event is reachable without any checks at all. We examined each report by hand and classified it as a false positive or real vulnerability.

Note the importance of role partitioning in Table 2 for reducing the number of false positives. Without role partitioning, a conservative analysis might assume that critical variables should be checked consistently in all program contexts. All contexts associated with roles that do not require a particular security check would then result in false positives.

The number of false positives after role-specific consistency analysis is very small. There are several reasons for the remaining false positives. First, if a role contains only one context, RoleCast cannot apply consistency analysis and conservatively reports a potential vulnerability. Second, a Web application may use a special set of critical variables only for a small fraction of contexts (this case is rare). Consider Figure 7. Both the `post2()` method call on Line 11 in index.php and the `fullNews()` method call on Line 4 in news.php contain security-sensitive events. A large fraction of calling contexts use `$auth` variable to enforce access control (Line 10 in auth.php). On the other hand, a small fraction of contexts leading to the sensitive database operation

	false positives		no	
Web applications	**roles**	**no roles**	**auth.**	**vuln.**
minibloggie 1.1	0	0	0	1
DNscript	1	5	0	3
mybloggie 2.1.6	0	0	0	1
FreeWebShop 2.2.9	0	1	0	0
Wheatblog 1.1	1	0	1	0
phpnews 1.3.0	1	12	0	0
Blog199j 1.9.9	0	1	0	0
eBlog 1.7	0	4	2	0
kaibb 1.0.2	0	11	1	0
JsForum (JSP) 0.1	0	0	0	5
JSPblog (JSP) 0.2	0	0	0	3
totals	3	34	4	13

Table 2: Accuracy ($\theta_{consistency}$ = .5). Note the reduction in false positives due to role partitioning.

in `fullNews` use only `$Settings` (Line 9 in news.php). Because RoleCast decides that `$auth` is the variable responsible for security enforcement due to its consistent presence in the contexts of security-sensitive events, it decides that the few contexts that only use `$Settings` are missing a proper security check.

Table 2 distinguishes between two kinds of unauthorized database operations. Some database updates may be relatively harmless, *e.g.*, updating counters. Nevertheless, if such an update is executed without a security check, it still enables a malicious user to subvert the intended semantics of the application. Therefore, we do not consider such updates as false positives and count them in the 3rd column of Table 2, labeled "no auth.". The 4th column of Table 2, labeled "vuln.," reports database updates that allow a malicious user to store content into the database without an access-control check. Because these vulnerabilities are both severe and remotely exploitable, we notified the authors of all affected applications.

Figure 6 shows two files from DNscript that RoleCast reports as vulnerable. Neither file contains any security checks, thus a malicious user can alter the contents of the back-end database by sending an HTTP request with the name of either file as part of the URL. RoleCast reports that the security-sensitive events in `DelCB.php` and `admin/AddCat2.php` should be protected by checking `$_SESSION[‘member’]` and `$_SESSION[‘admin’]`, respectively.

Our analysis uses three thresholds: the branch asymmetry threshold θ_{asymm}, the commonality threshold for seed files θ_{seed}, and the consistency threshold for security-critical variables $\theta_{consistency}$. Tables 3 through 5 show that the analysis is not very sensitive to these values.

Table 3 shows the sensitivity of our results to the branch asymmetry threshold θ_{asymm} as it varies between 25 and 200. The default value is 100. With θ_{asymm} values smaller

	θ_{asymm}									
	25		50		100		150		200	
Web applications	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**
minibloggie 1.1	1	0	1	0	1	0	1	0	1	0
DNscript	3	1	3	1	3	1	3	1	3	1
mybloggie 1.0.0	1	0	1	0	1	0	1	0	1	0
FreeWebShop 2.2.9	0	0	0	0	0	0	0	0	0	0
Wheatblog 1.1	1	1	1	1	1	1	0	0	0	0
phpnews 1.3.0	0	1	0	1	0	1	0	1	0	1
Blog199j 1.9.9	0	2	0	1	0	0	0	0	0	0
eBlog 1.7	2	0	2	0	2	0	2	0	1	0
kaibb 1.0.2	1	0	1	0	1	0	1	0	1	0
JsForum (JSP) 0.1	3	0	3	0	3	0	3	0	3	0
JSPblog (JSP) 0.2	5	0	5	0	5	0	5	0	5	0

Table 3: Sensitivity of actual vulnerabilities (vl) and false positives (fp) to θ_{asymm}

	θ_{seed}									
	0.2		0.3		0.4		0.5		0.6	
Web applications	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**
minibloggie 1.1	1	0	1	0	1	0	1	0	1	0
DNscript	3	1	3	1	3	2	3	1	3	1
mybloggie 1.0.0	1	0	1	0	1	0	1	0	1	0
FreeWebShop 2.2.9	0	0	0	0	0	0	0	0	0	0
Wheatblog 1.1	1	1	1	1	1	1	1	1	1	1
phpnews 1.3.0	0	1	0	1	0	1	0	1	0	1
Blog199j 1.9.9	0	0	0	0	0	0	0	0	0	0
eBlog 1.7	2	0	2	0	2	0	2	0	2	0
kaibb 1.0.2	1	0	1	0	1	0	1	0	1	11
JsForum (JSP) 0.1	3	0	3	0	3	0	3	0	3	0
JSPblog (JSP) 0.2	5	0	5	0	5	0	5	0	5	0

Table 4: Sensitivity of actual vulnerabilities (vl) and false positives (fp) to θ_{seed}

	$\theta_{consistency}$									
	0.5		0.6		0.7		0.8		0.9	
Web applications	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**
minibloggie 1.1	1	0	1	0	1	0	1	0	1	0
DNscript	3	1	3	1	3	1	3	0	0	0
mybloggie 1.0.0	1	0	1	0	1	0	1	0	1	0
FreeWebShop 2.2.9	0	0	0	0	0	0	0	0	0	0
Wheatblog 1.1	1	1	1	1	1	1	1	1	1	1
phpnews 1.3.0	0	1	0	1	0	1	0	1	0	1
Blog199j 1.9.9	0	0	0	0	0	0	0	0	0	0
eBlog 1.7	2	0	2	0	1	0	1	0	1	0
kaibb 1.0.2	1	0	1	0	1	0	1	0	1	0
JsForum (JSP) 0.1	3	0	3	0	3	0	3	0	3	0
JSPblog (JSP) 0.2	5	0	5	0	5	0	5	0	5	0

Table 5: Sensitivity of actual vulnerabilities (vl) and false positives (fp) to $\theta_{consistency}$

admin/AddCat2.php

```php
<?php
// No security check. It should have been checked
      with $_SESSION[‘admin’]
include ’inc/config.php’;
include ’inc/conn.php’;
$values = ’VALUES (”’.$_POST[’cat_name’].’”)’;
// Security−sensitive event
$insert = mysql_query(”INSERT INTO gen_cat(cat_name)
      ” .$values);
if($insert)
{
  mysql_close($conn);
  ...
}

?>
```

DelCB.php

```php
<?php
// No security check. It should have been checked
      with $_SESSION[‘member’]
include ’inc/config.php’;
include ’inc/conn.php’;
// Security−sensitive event
$delete = mysql_query(”DELETE FROM close_bid where
      item_name = ’”.$item_name.”’”);
if($delete)
{
  mysql_close($conn);
  ...
}
?>
```

Figure 6: Detected vulnerable files in DNscript

index.php

```php
if ($_GET[‘action’] == ‘redirect’)
{
  ...
}
$time_start = getMicrotime();
define(‘PHPNews’, 1);
session_start();
require(‘auth.php’);
...
// Security−sensitive operation is in post2
post2();
```

auth.php

```php
session_start();
...
$result = mysql_query(’SELECT * FROM ’.$db_prefix.’
      posters WHERE username = \’’.$in_user.’\’ AND
      password = password(\’’.$in_password.’\’)’);
$dbQueries++;
if(mysql_numrows($result) != 0)
{
  $auth = true;
  ...
  // Security check using critical variable $auth
  if(!$auth) {
    exit;
  }
}
```

news.php

```php
include(’settings.php’);
...
else if($_GET[’action’] == ’post’)
  fullNews();
...
function fullNews(){
  ...
  // Critical variable $Settings
  if($Settings[’enablecountviews’] == ’1’) {
    $countviews = mysql_query(”UPDATE ”.$db_prefix.”
        news SET views=views+1 WHERE id=’”.$_GET[’id’
        ].”’”);
  }
   ...
}
```

Figure 7: Example of a false positive in phpnews 1.3.0

than 100, ROLECAST analyzes more branches, more critical methods, and more critical variables, but still finds the same vulnerabilities, although with two more false positives when $\theta_{asymm} = 25$.

Table 4 shows the sensitivity of our results to the value of the seed threshold θ_{seed} used to generate role partitions. The default value is 0.2. For *kaibb*, when θ_{seed} is too large, ROLECAST may exclude seed files that actually play an important role in partitioning the application into roles. Note that the results for DNscript do not change monotonically with the value of θ_{seed}. When $\theta_{seed} = .2$ or .3, ROLECAST finds three seed files. Two of them correspond to actual user roles (administrator and regular user) and ROLECAST produces the correct partition. When $\theta_{seed} = .4$, there are only two seed files, one of which corresponds to the user role, while the other produces a spurious “role” with a single context, resulting in a false positive. When $\theta_{seed} = .5$ or .6, ROLECAST finds a single seed file, which corresponds to the administrator role. The resulting partition has two groups—contexts that use the seed file and contexts that do not use the seed file—which are exactly the same as in the partition created when $\theta_{seed} = .2$ or .3.

Table 5 shows how our results change as the $\theta_{consistency}$ threshold varies between 0.5 and 0.9. This threshold controls the fraction of critical-file contexts in which a variable must appear in order to be considered security-critical for the given role. The default value is 0.5. In two applications, increasing the threshold decreases both the number of vulnerabilities detected and the number of false positives (as expected). For most applications, there is no difference because the root cause of many reported vulnerabilities is either the absence of any checks prior to some security-sensitive event, or roles containing a single context.

In summary, the algorithm is not sensitive to its three thresholds, requires role analysis to reduce the false positive rate, and finds actual vulnerabilities.

10. Conclusion

We designed and implemented ROLECAST, a new tool for statically finding missing security checks in the source code of Web applications without an explicit policy specification. Prior approaches to static, "specification-less" verification of security mediation assumed that security checks are syntactically recognizable and/or that the same pattern of security checks must be used consistently in all applications. Neither approach works for Web applications because different applications and even different roles within the same application use different, idiosyncratic security checks.

ROLECAST exploits the standard software engineering conventions used in server-side Web programming to (1) identify security-sensitive operations such as database updates, (2) automatically partition all contexts in which such operations are executed into groups approximating application-specific user roles, (3) identify application- and role-specific security checks by their semantic function in the application (namely, these checks control reachability of security-sensitive operations and a failed check results in quickly terminating or restarting the application), and (4) find missing checks by consistency analysis of critical variables within each role.

When evaluated on a representative sample of open-source, relatively large PHP and JSP applications, ROLECAST discovered 13 previously unreported vulnerabilities with only 3 false positives.

Acknowledgments

The research described in this paper was partially supported by the NSF grants CNS-0746888, CNS-0905602, and SHF-0910818, a Google research award, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of Web-based applications. In *CCS*, pages 25–35, 2007.

[2] M. Bond, V. Srivastava, K. McKinley, and V. Shmatikov. Efficient, context-sensitive detection of real-world semantic attacks. In *PLAS*, pages 1–10, 2010.

[3] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS*, pages 39–50, 2008.

[4] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *PLDI*, pages 234–245, 2011.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[6] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication and access control vulnerabilities in Web applications. In *USENIX Security*, pages 267–282, 2009.

[7] D. Denning and P. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–513, 1977.

[8] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in Web applications. In *USENIX Security*, pages 143–160, 2010.

[9] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing Web application code by static analysis and runtime protection. In *WWW*, pages 40–52, 2004.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *S&P*, pages 258–263, 2006.

[11] JSP. http://java.sun.com/products/jsp.

[12] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *OOPSLA*, pages 359–372, 2002.

[13] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, pages 75–86, 2009.

[14] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *ESEC/FSE*, pages 296–305, 2005.

[15] PHP. http://www.php.net.

[16] PHP advent 2010: Usage statistics. http://phpadvent.org/2010/usage-statistics-by-ilia-alshanetsky.

[17] M. Pistoia, R. Flynn, L. Koved, and V. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, pages 362–386, 2005.

[18] Quercus. http://quercus.caucho.com.

[19] A. Sistla, V. Venkatakrishnan, M. Zhou, and H. Branske. CMV: Automatic verification of complete mediation for Java Virtual Machines. In *ASIACCS*, pages 100–111, 2008.

[20] S. Son and V. Shmatikov. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *PLAS*, 2011.

[21] Soot: A Java optimization framework. http://www.sable.mcgill.ca/soot/.

[22] V. Srivastava, M. Bond, K. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *PLDI*, pages 343–354, 2011.

[23] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *USENIX Security*, pages 379–394, 2008.

[24] Apache Tomcat. http://tomcat.apache.org.

[25] G. Wasserman and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, pages 32–41, 2007.

[26] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, pages 179–192, 2006.

[27] A. Yip, X. Wang, N. Zeldovich, and F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.

A. Building the Call Graph

Security analysis performed by ROLECAST requires a precise call graph. As mentioned in Section 3.2, we first translate Web applications into Java. For JSP, this translation produces well-formed method invocations and we construct an accurate call graph for the resulting Java code using the class hierarchy analysis (CHA) in Soot. For PHP, however, the translation performed by the Quercus compiler makes all method calls indirect, either via reflective calls or via lookups in a hash table. The CHA analysis in Soot does not support either and thus cannot be used directly to construct the call graph of the resulting Java program.

Quercus translates each PHP function into a distinct Java class. Therefore, to statically find the targets of unresolved method calls, it is sufficient to perform intraprocedural propagation of symbolic method names in the main method of the Quercus-generated class. First, ROLECAST analyzes the initialization methods, which assign to the global method lookup hash table. It initializes each member variable in the class to the symbolic method name(s) in the initialization methods. ROLECAST then propagates symbolic names within the main function of each class until it reaches a fixed point. Reflective calls dynamically reference an actual argument to resolve their target methods. Therefore, ROLECAST also checks symbolic values of arguments at unresolved callsites. ROLECAST then adds a call-graph edge between the callsite and the target method.

The call graph constructed by ROLECAST using this technique is object- and context-insensitive. As described in Section 5.1, ROLECAST then uses a *context-sensitive* algorithm to compute the contexts in which security-sensitive operations may be executed. In our experimental evaluation, ROLECAST resolved 95% or more of user-defined method calls in every benchmark Web application. The remaining unresolved callsites were all due to missing plug-in function bodies. Only two callsites in our benchmarks required manual annotation.

Recall that in Web applications, each program file can be invoked directly by the network user and thus represents a potential entry point. These entry points require additional edges in the call graph. The call-graph analysis in ROLECAST identifies all potential program entry points and constructs a single call graph with multiple entry points—one entry point for each program file and all methods it contains.

However, programmers often neglect to defensively program unintended entry points. This creates calling sequences in the call graph that include methods not defined in the current file, since the programmer intended this file to be invoked only from some other file which does define these methods. The PHP interpreter will stop execution if the program invokes an undefined method.

In theory, we simply need to eliminate all calls to undefined methods in the call graph. In our current implementation, we perform this pruning of undefined methods on the set of calling contexts CC for each security-sensitive event e. We compute the dominator relationship for every method. For each method $m \in cc \in CC$, if it dominates e interprocedurally but m is undefined, we eliminate the calling context cc from the set CC. Eliminating undefined methods in the call graph should produce the same result as eliminating them in the set of calling contexts, and the efficiency of our analysis can be improved by performing this elimination directly on the call graph.